Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Engineering 24 (2011) 171 – 176

Procedia Engineering

www.elsevier.com/locate/procedia

2011 International Conference on Advances in Engineering

A Distributed Real-time Database Index Algorithm Based on B+ Tree and Consistent Hashing

Xianhui Li[a], Cuihua Ren[b], Menglong Yue[a,c,a]*

[a]*China Realtime Database CO.LTD., SGEPRI, 210000, China*
[b]*China Communications 2nd Navigational Bureau 2nd Engineering Co., Ltd., Chongqing, 404100, China)*
[c]*Software Institute of Nanjing University, Nanjing, 21000, China*

Abstract

This paper proposed a novel method of Distributed real-time database index algorithm based on B+ Tree and consistent hash. In order to determine the storage location of each TAG point in the distributed environment, First of all, every storage node and each TAG point are mapped to circular hash space. Secondly, create a hash table of TAG point in every storage node, which record the position of index in every TAG point. Finally, a B+ Tree index are established to organize and maintain the historical data of one TAG point. Theoretical analysis and experimental results show the validity of the proposed method.

Keywords: Distributed System, Real-time Database, Hiberarchy Index, Consistent Hashing;

1. Introduction

With the development of computer technology and enhancement of Automation technology, there has been a lot of data access and management applications with time constraints, such as power system scheduling, industrial control, securities trading, aerospace, and so on. These applications often require real-time sampling of the monitoring equipment to understand the system real-time operation status, which with a very high acquisition frequency, such as 25 per second, 50 per second or even 100 per second. At the same time, all the data within the specified time must be saved completely, thus the need to maintain huge amounts of data. Also it calls for Data acquisition, process and make the right response within a designated time or time range, with a significant time-sensitive. There are so massive, real-time, high-frequency data that the traditional relational database is hard to meet the needs of the application, whether to store or retrieve. In recent years, with the emergence of real-time database, it is possible to

*Corresponding author. Tel.:13951687315.
E-mail address: lixianhui@sgepri.sgcc.com.cn.

1877-7058 © 2011 Published by Elsevier Ltd. Open access under CC BY-NC-ND license.
doi:10.1016/j.proeng.2011.11.2621

implement the functions of these applications. And now real-time database has become a research hotspot [1]. Currently, there are some mature real-time database system at home and abroad, including the OSIsoft's PI[2] and InStep's eDNA[3] in the United States, HighSoon [4] and LiRTDB [5] real-time database in China.

A real-time database is specially designed to deal with the data with a characteristic of time sequence of database management system, which is used for the storage and management of the real-time, high frequency and massive data above mentioned. At the same time, in order to improve the system scalability, fault tolerance and retrieval speed, it is necessary to make the real-time database distributed, that's to say a distributed real-time database system is necessary. Just because of the characteristic of real-time, high frequency, massive and distributed of distributed real-time database system, to get a better index method is playing a crucial role for efficiently storing and retrieval. Based on this objective, this paper puts forward a distributed real-time database Hiberarchy index algorithm, first of all, we using the consistency hash algorithm to make sure the corresponding relationships between TAGs and storage nodes. Then, take the TAG name or ID as hash key value, we record the TAGs in each storage node with a hash table to maintain the TAGs belong to it. Finally, Construct a B+ Tree for each TAG to index all the data of the TAG. By comparing several index methods in Experimental section, it shows the validity of the proposed method.

2. Distributed real-time database framework

There are two type of node in the distributed real-time database system[6][7][8], one is the center control server named NameServer, which can exit only one in the whole system. It is used to storage the related metadata of the whole system, such as the data storage server information, data parting information, access control information and so on. Another is the data storage server named DataServer, which can exit one or several in the whole system. And also it could be built in different computer. This type of node is mainly used to data storage in distributed real-time database.

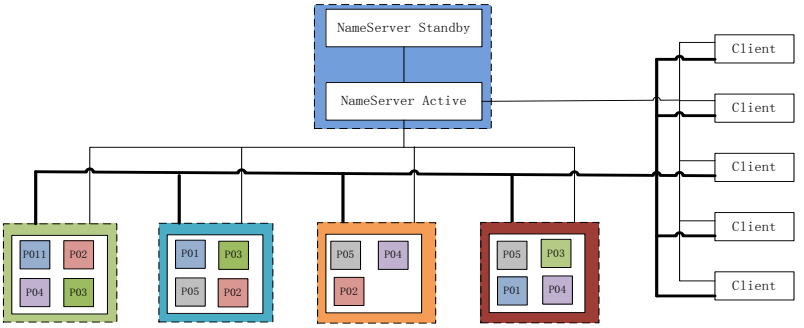

Fig.1 Distributed real-time database framework diagram

When the client wants to storage or retrieval data, first of all, it sends a request to the NameServer to inquire the location of the actual data. And then communicate with the actual DataServer to do the really data storage or retrieval. That's to say that actual data transmission is between Client and DataServer. In order to improve the availability and reliability, we erect the NameServer with Dual-Computer Hot-Standby. Normally, NameServer active provide service. Once NameServer Active with a fault occurs and stops provide service, NameServer Standby will automatically switch to Active mode and providing service to ensure system availability and reliability. Taking TAG as unit, DataServer storage lots of data files. In order to improve the system's usability and fault tolerance, each data file in the whole distributed real-time database having many copies. At the same time TAG is the unit of the dynamic load balancing. Through the analysis of the dynamic load of each DataServer, NameServer dynamic adjustment the load

in the whole distributed real-time database system. With the heartbeat mechanism, NameServer get the operation status of each DataServer. The Heartbeat package, which contains DataServer's CPU, memory, and disk usage, is the basis for dynamic load balancing. Figure 1 is a typical case of distributed real-time database framework.

3. Hierarchy index

3.1. Data partition

Along with the increasing amount of data in distributed real-time database systems, how to better storage and management the increasing data become the main index of distributed real-time database performance. A better method is to part the data of the system [9]. To meet the performance of the system requirements, the whole system data will share in many DataServer through the data partition, which make the data quantity be much smaller in every DataServer. Certainly, there are many kinds of method to part the whole system data. We can part data with TAG ID. Of course time range is a good choice. And someone take data quantity as the division standard. In this paper, we take TAG ID as the division standard. In order to improve the system fault tolerance and minimize the node online or node offline, which will trigger to rehash, and then a large number of data will migrate among the whole system DataServer. Combined with the company business, we choose the hash algorithm proposed in literature [10] [11] [12]. With this method, the remove/add a data nodes always, it can be as small as possible to change the already exist key mapping relation among the DataServer to meet the requirements of monotonicity, balance and spread. The steps of the data partition method proposed in this paper as follows:

3.2. To construct hash space

A value into an n-bit key, 0~2^n-1. Now imagine mapping the range into a circle, then the key will be wrapped, and 0 will be followed by 2^n-1. In this paper, we take n for 32. Then the hash space will as show in figure 2. (a) And we take the map function as:

Key = hash(objectID);

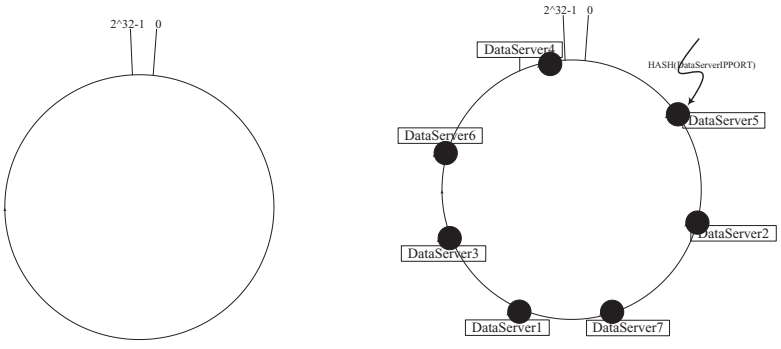

Fig.2 (a) hash space; (b) The Distribution of DataServer after mapping

3.2.1. Map DataServer into hash space

In the system initialization process, we use a hash function to get all the DataServers key values and map them into the hash space. In this paper, we assume that there are seven DataServers existing in the whole system. After the initialization process, those DataServers are distributed in the hash space as show in Figure 2. (b).

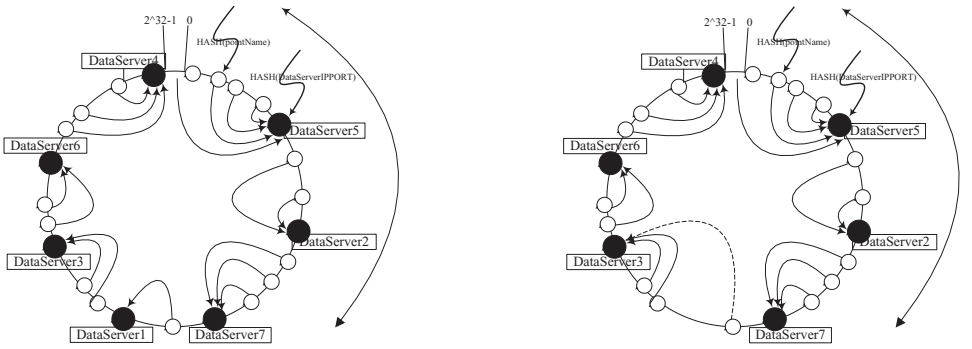

Fig.3 (a) the Key value distribution of DataServer and TAG point after mapping; (b) TAG the Key value distribution after DataServer failure

3.2.2. Map TAG point into DataServer

In the process of adding TAG point, client sends request to NameServer. NameServer calculated the MD5 value of this TAG point according to request TAG point's features identification code (such as point name, point ID). And then map the MD5 value to the hash space with the same hash algorithm, looking for DataServer in clockwise direction (hash key values increase direction), and the first found DataServer is where this TAG point data will be storage. In this paper, we suppose the whole system exist 17TAG points(P1 ~ P17), then after the step of map DataServer into hash space, the distribution of those TAG point in the hash space as shown in fig.3. (a) And the distribution of TAG points has shown in table 1.

Table1 TAG point store in each DataServer

DataServer	TAG point	DataServer	TAG point
DataServer1	P10	DataServer5	P02,P09,P14,P16
DataServer2	P04,P12	DataServer6	P06,P11
DataServer3	P01,P07	DataServer7	P05,P08,P15
DataServer4	P03,P12,P17		

3.2.3. After dDataServer failure over

With the consistent hashing algorithm, when a new DataServer join in or the existing DataServer failure off, we ensure that nothing should to do with that except migrate the failure DataServer data to the other exist DataServer in hash space. As shown in figure 3 (b), when DataServer1 failure off, we just need to migrate the data of DataServer1 to DataServer3, the other components shouldn't be changed.

When the client wants to insert or query data, firstly, it sends request to NameServer to get where the TAG point is. This is the first step of our hierarchy Index method: make sure which DataServer storage the requested TAG point's data.

3.3. TAG point Index

There is a hash table names PointHashTable In each DataServer internal, which record the detailed information of every TAG point in this DataServer. The detailed point information include: point name, point ID, the index location, that to say the root node location of the tag point B+ Tree, etc. PointHashTable realize like that:

map(int PointID, PointConfigItem* item).

In PointConfigItem structure, rtCache point to the corresponding Cache of TAG point. In Cache structure, there is a pointer rawHist, pointing to the root node of B+ Tree. After get the TAG point's

storage DataServer, the client then communicates with that DataServer. If requested to add point, we map the TAG point detailed information to the appropriate slot of the hash table PointHashTable by hashing with the point characteristics identification code, and then storage the PointConfigItem of that point to the PointHashTable. While if requested to insert value, DataServer calculate the hash value of this point by using the TAG point name or TAG point ID, and get the PointConfigItem from PointHashTable. Then we can get the B+ Tree root node location and traverse B+ Tree to find where the insert data storage in. This is the second step of our hierarchy Index method: make sure the position of TAG point index. The PointHashTable shows as figure 4.

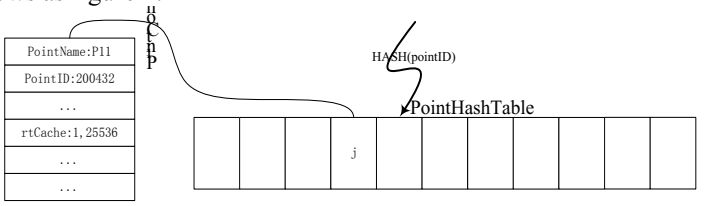

Fig.4 DataServer hash table with TAG points

3.4. Data index

After get the position of TAG point index. The B+ Tree is traversed from roots node if store or retrieve data requested. And then compare the requested data time range with the B+ Tree node. If time range match, then traversed the child node until to the leaf node. And this leaf node is the node which the requested data insert in or storage according to the request type, storage or insert data. This is the third step of our hierarchy Index method: To determine where to get or put the request data.

In the DataNode structure of B+ Tree, we make some changes. To link all of the DataNode in the same B+ Tree with prev and next pointers, and make it like a doubly linked list with which it can increase the speed of batch retrieval. Each TAG point's B+ Tree index structure shows in Figure 5.

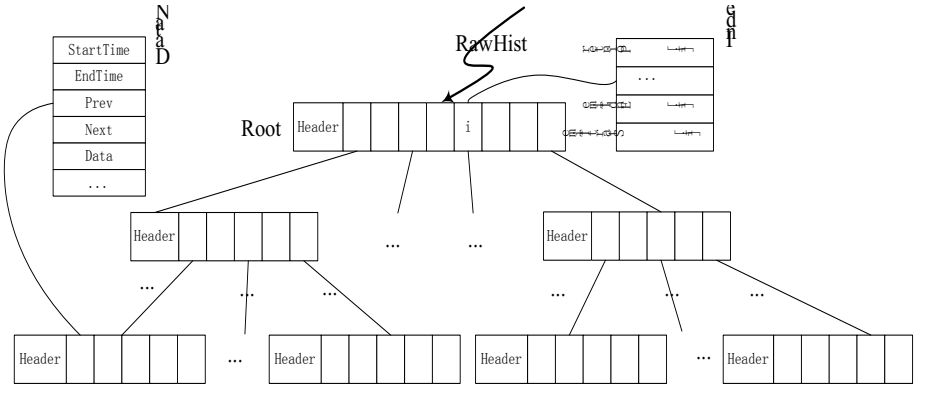

Fig.5 B+ Tree index structure diagram in DataServer side of distributed real-time database

4. Experimental results and analysis

In this section, we compare the insert and query efficiency among three different index methods in the platform of HighSoon, which is the main product of China Realtime Database CO.LTD. For superiority of partition, we can refer to [6-10]. This experiment focused on get the insert and retrieval efficiency among different data index method. Comparison of insert and retrieve performance among B+ Tree, RB- tree and T- tree. The test platform is based on points of 20 million TAG points; insert 10 million of events to each

TAG point. The results shown in Table 2, the unit for the million events per second, can be seen from the table, B+ Tree as a large amount of data in the persistent system has better performance.

Table 2 Insert and retrieve performance comparison among four data interpolation index structure

Function	B+ Tree	Red-Black Tree	T- Tree
Batch insert	309.2	221.5	210.2
Cross section insert	160.0	120.1	100.6
query	119.6	91.8	49.7

5. Conclusion

This paper presents a new data index method of distributed real-time database. Through the introduction of the Consistent hashing algorithm, we can determine the real-time database data fragmentation rules in distributed environment. Using of hashtables and B+ Tree index method, we can maintain the data for each TAG point. Experimental comparison of the proposed method with some common methods between insert and retrieval efficiency, and the proposed method is proved better. As the company's main market is the power system, the next step is mainly to do parameter tuning on the index to adapt to different industries insert and retrieval efficiency requirements.

References

[1].Ben Kao, Hector Garcia-Molina. An overview of real-time database systems[R]: Tech. Report of Princeton University, Stanford University 1990.

[2].OSI,PI_System_Standards[EB/OL].http://www.osisoft.com/software-support/what-is-pi/PI_System_Standards.aspx.

[3].InStep, edna_overview[EB/OL]. http://www.instepsoftware.com/edna_overview.asp .

[4].CRD ,HighSoon.[EB/OL] .http://crd.sgepri.sgcc.com.cn/html/cp68.shtm HighSoon.

[5].LUCULENT ,LiRTDB[EB/OL] .http://www.luculent.net/project/project-sssjk.asp LiRTDB.

[6].Fay Chang, Jeffrey Dean, Sanjay Ghemawat, etc. Bigtable: A Distributed Storage System for Structured Data[J]. Journal of ACM Transactions on Computer Systems (TOCS). TOCS Homepage archive Volume 26 Issue 2, June 2008, ACM New York, USA.

[7].Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. The Google file system[J]. Proceeding of SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles, Volume 37 Issue 5, December 2003, ACM New York, NY, USA.

[8].Jeffrey Dean, Sanjay Ghemawat. MapReduce: simplified data processing on large clusters[C]. Communications of the ACM - 50th anniversary issue: 1958 – 2008 CACM Homepage archive. Volume 51 Issue 1, January 2008. ACM New York, NY, USA.

[9].Wikipedia, Shard(database architecture)[EB/OL]. http://en.wikipedia.org/wiki/Shard_(database_architecture).

[10].David Karger, Eric Lehman, etc. Consistent Hashing and Random Trees: Distributed Caching Protocls for Relieving Hot Spots on the World Wide Web[C]: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, New York, 1997[C].

[11].Giuseppe Decandia, Deniz Hastorun, Madan Jampani, etc. Dynamo: amazon's highly available key-value store[C]. Proceeding of twenty-first ACM SIGOPS symposium on Operating systems principles, V.41 Issue 6, 2007. ACM New York, USA.

[12].THE CODE PROJECT, Consistent hashing [CP]. http://www.codeproject.com/KB/recipes/lib-conhash.aspx.